AFRL-IF-RS-TR-2004-210
Final Technical Report
July 2004

COPING WITH COMPLEXITY: A STANDARDS-BASED KINESTHETIC APPROACH TO MONITORING NON-STANDARD COMPONENT-BASED SYSTEMS

Columbia University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K503

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

AFRL-IF-RS-TR-2004-210 has been reviewed and is approved for publication

APPROVED: /s/

JAMES R. MILLIGAN
Project Engineer

FOR THE DIRECTOR: /s/

JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** JULY 2004	**3. REPORT TYPE AND DATES COVERED** Final Jun 00 – Dec 03

4. TITLE AND SUBTITLE COPING WITH COMPLEXITY: A STANDARDS-BASED KINESTHETIC APPROACH TO MONITORING NON-STANDARD COMPONENT-BASED SYSTEM	**5. FUNDING NUMBERS** C - F30602-00-2-0611 PE - 62302E PR - DASA TA - 00 WU - 02
6. AUTHOR(S) Gail E. Kaiser	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Columbia University The Trustees 1210 Amsterdam Avenue MC 2205 New York New York 10027	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency, 3701 North Fairfax Drive, Arlington Virginia 22203-1714; AFRL/IFTB, 525 Brooks Road, Rome New York 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2004-210
11. SUPPLEMENTARY NOTES AFRL Project Engineer: James R. Milligan/IFTB/(315) 330-1491/ James.Milligan@rl.af.mil	
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

Technology was developed to support runtime monitoring – i.e., continual validation – regarding the dynamic functional and extra-functional properties of component-based systems. Software probes are inserted or wrapped into component ports and actualized connector middleware, to report system events that cross/impact component and connector boundaries. Required an prohibited properties are defined as (potentially complex) patterns over collections of events. The monitoring infrastrutre includes recognizers to detect the orrurrence or omission of these patters as the components and connectors inresponse to monitored activities.

14. SUBJECT TERMS Autonomic Computing, Component-Based Software Engineering, OODA Feedback Control Loop, Software Architecture, Software Configuration	**15. NUMBER OF PAGES** 45
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

1.0 Executive Summary

The increasing complexity of networked computer systems and applications has led to a tremendous interest in what some have termed *autonomic computing*: the notion of self-managing software as an attractive approach to reducing the time and effort costs of operating and maintaining software systems, and to increasing their dependability and assurance levels. Some such technologies are already being promoted commercially, e.g., several major vendors sell enterprise applications that require little help from IT staff to run and maintain. However, most ignore both legacy software and the increasingly common assembly of large scale systems from components supplied by multiple sources, including COTS, GOTS and open source, instead assuming the customer or user will be willing and able to adopt a new generation of "solutions" from a single vendor.

A few general-purpose facilities have been developed to automate problem detection for arbitrary target systems. For example, some new operating systems include engines to automate the collection of crash data; other tools help detect anomalous behavior by monitoring system and application logs. However, these approaches generally leave analysis of what the system is doing (or not doing), how and why, to a human systems administrator, who must then determine, plan and carry out the reconfiguration or repair.

To do better, Columbia has investigated a generic framework for not just collecting but also *interpreting* behavioral data at runtime, said interpretation tailored to the target system by introduction of system models that describe expected correct behaviors and possibly also anticipated error situations that can automatically be recognized as having occurred, or not occurred, as the case may be. Further, the framework includes a *feedback control loop* to automatically decide when corrections are required, construct and instantiate repair plans, and coordinate the performance (and handle contingencies) of the possibly many interdependent elements required for target system reconfiguration – in many cases with no "down time", while the system continues operation.

Columbia has developed the "Kinesthetics eXtreme" (KX, pronounced "kicks") autonomic computing infrastructure for legacy systems and systems of systems, consisting of four main kinds of components - sensors, gauges, controllers, and effectors – communicating with each other through a publish/subscribe event notification facility. KX is the only complete implementation of the reference architecture (see Figure 1), and standard APIs for these components, devised together with a consortium of DASADA researchers (including Teknowledge, CMU, OBJS and BBN as well as Columbia and its subcontractor WPI).

Sensors (also known as *probes*) watch the target system to collect primitive data, while separate *gauges* aggregate, filter and interpret the sensor data according to system models. KX's sensor/gauge monitoring framework can be used with or without a control feedback loop that automatically performs dynamic adaptations. Without the feedback loop, gauges would typically generate alerts and/or be visualized on a human systems management console.

KX's automated adaptation framework supplements the monitoring framework with decision, coordination and actuation capabilities. Based on the coalesced and interpreted sensor data relayed by gauges and on modeled information about the target system, a *controller* makes decisions on what adaptations (if any) need to be done. This triggers a controller facility to plan and orchestrate the work of one or more *effectors* - which interact with the target system to carry out the low-level tweaks and tuning, and/or coarser subsystem restarts and reconfigurations, directed by the adaptation plan.

KX has been used successfully to monitor and repair a variety of target systems - including DoD, industry, and academic applications - thus demonstrating achievement of the self-configuring, self-healing, self-optimizing goals of autonomic computing for legacy systems and systems of systems.

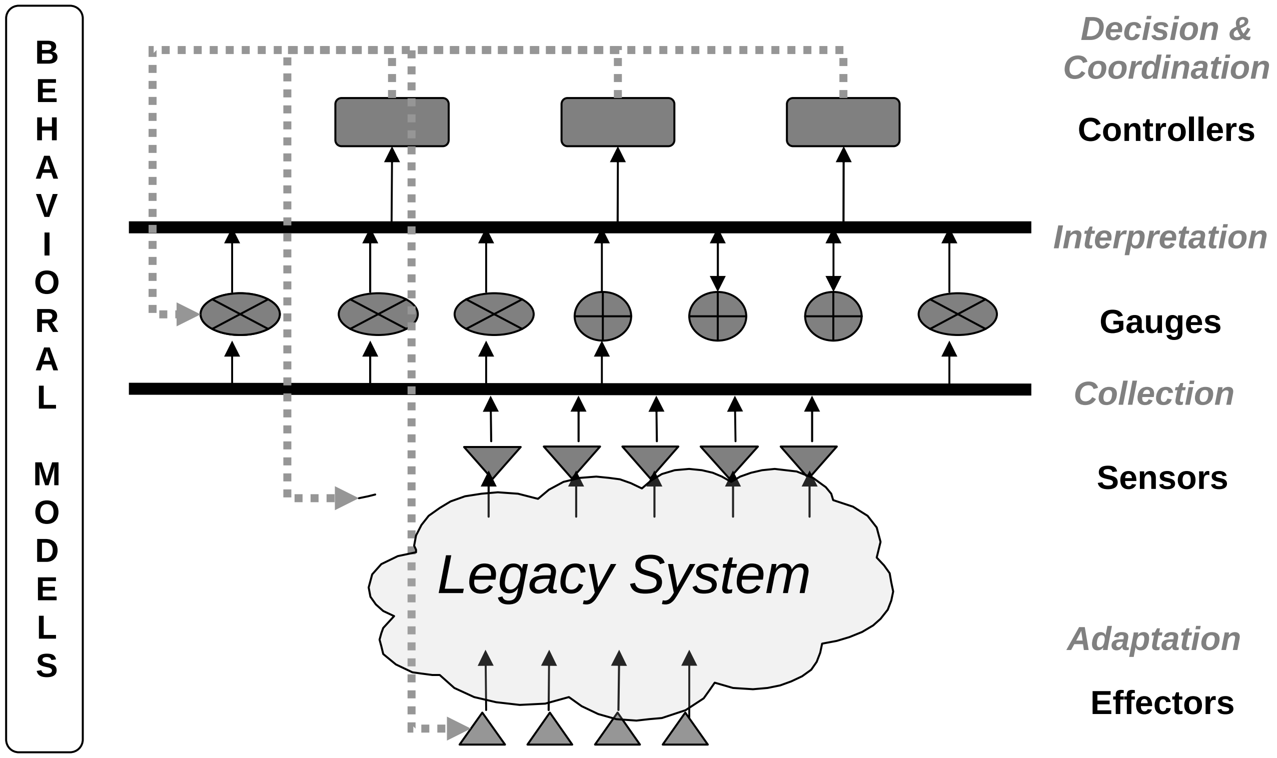


Figure 1. Reference Architecture

2.0 Objectives and Scope

Defense applications are rightfully viewed as fundamentally different from mass-market general-purpose software. There are relatively few customers and it is unacceptable for critical defense applications to suddenly terminate with a message asking whether or not the user wants to submit the error data to Microsoft, as is (unfortunately) typical of many consumer software packages. Concerns about security have led in recent years to some "hardening" of enterprise server software, but it is still common to solve glitches in many data centers by restarting the applications or rebooting the host (or restarting on another host), typically conducted manually by systems administration staff. However, the DoD requires its critical software systems to achieve the highest levels of assurance, dependability, and adaptability – and often continuous availability and survivability: human systems management, alone, is usually not an acceptable solution.

The DASADA program presented a paradigm shift offering a novel and promising approach, intended to enable the DoD to adopt and leverage the proven functionality of existing COTS and open source software assets: Continual design, continual coordination and continual validation. In this model, components are selected or constructed, customized to the application domain, and continuously evaluated before, during and after system assembly - and on-the-fly reassembly - to ensure that they can and do operate together with the rest of the system, and its current context, within the tolerated bounds. Continual monitoring and reconfiguration (validation and coordination, respectively) are particularly essential for assured applications, since assurances that may have been met at initial system design time may not prove to be appropriate for field conditions that may be subject to rapid change while the system is running. Such applications typically cannot be "taken down" for long re-engineering or enhancement cycles, but must be dynamically and automatically repaired in response to runtime sensor data and feedback from runtime analysis gauges of functional and extra-functional system properties.

Two necessary bases for all three facets of dynamic assembly are: (1) being able to precisely determine and usefully specify the variation inherent in using "foreign" software components, connectors, and their configurations; and (2) being able to measure that software components, including middleware and other actualized connectors, fit and continue to fit together as system and context change, within functional and extra-functional tolerances permitted by the dynamically evolving requirements of critical software systems. The measurement probes must be insertable into legacy as well as new components and compositions, and presentable to humans and automated decision algorithms in timely fashion as readily interpretable gauges, to prevent inappropriate system assemblies and trigger reassemblies promptly when needed.

However, it is simply impossible to include all possible meters, and appropriate mechanisms for interpreting gauge notifications, say, as customizations built into a system when it is designed or initially deployed: New components and platforms, or advancing technologies, may mandate or enable new metrics and feedback (or feedforward). Thus a practical means is needed for rapidly inserting and configuring new sensors and gauges, and for deploying new repair plans, without significantly degrading system performance. Moreover, of course, measurement tasks and data must not themselves compromise assurance, dependability, adaptability, availability or survivability properties. One implication is that such data and tasks should be accessed and applied only on a "need to know" basis, rather than swamping communication, computation and storage resources by dispersing every sensor event throughout the distributed architecture – without knowledge of where it is "needed", e.g., to maintain a particular analysis gauge and/or instantiate a particular repair facility.

A model analogous to the DASADA paradigm has been promoted by IBM and its partners in recent years under the buzz phrase "autonomic computing", which has been widely adopted by the research and development community. However, most academic and industry efforts in this new field are primarily concerned with data center operations and/or grid computing, and – most significantly - assume that new software systems are designed and implemented from the ground up with autonomic capabilities (self-management, self-monitoring, self-reconfiguration, self-optimization, etc.) "built in". Although that approach may be ideal in some sense, it is impractical for either DoD or commercial applications involving legacy systems or systems of systems constructed out of components from multiple sources.

Columbia University and Worcester Polytechnic Institute (WPI) jointly proposed to the DASADA program to develop continual validation components and infrastructure in a form able to *retrofit* onto legacy systems and systems of systems *externally*, without any need to understand or modify the target system's code. Said continual validation functions as the open-loop frontend of autonomic computing, via intelligent automated monitoring of the target system. Although this was not explicitly part of the original proposal, Columbia also closed the autonomic feedback loop by developing backend components that perform automatic reconfiguration and/or repair, in essence adding continual coordination to the original continual validation goals. The lightweight design and separation of concerns eases adoption of individual components, whether directed to monitoring or repair, independent of the rest of the full infrastructure. The Columbia/WPI effort did not address the continual design aspect of DASADA.

3.0 Approach

The main components of this "external" infrastructure include sensors, gauges, controllers and effectors:

Sensors (or *probes* in DASADA parlance), watch target environment elements to produce time series of data. Sensors may emit relatively simple events corresponding to local activities as they occur, more complex events reflecting substantive summaries of logs, or alerts generated via an internal analysis (e.g., as in intrusion detection anomaly detectors). Most sensors are small, constrained, noninvasive pieces of code which get installed in or around the target system. There are numerous approaches to software system instrumentation, some of them available commercially. Some inject callbacks into source code (when source code is available and recompilation feasible), such as WPI's AIDE. Others modify bytecodes or binaries, such as OBJS' ProbeMeister, or replace Dynamic Linking Libraries (DLLs) or other dynamic libraries, such as Teknowledge's mediated connectors. Some operate in the surrounding environment, e.g., to inspect network traffic or operating system resource usage.

Gauges, which interpret and analyze data originating from one or more sensors. Gauges are intended to recognize that abstract semantic conditions or incidents have occurred (or are about to occur) in the target environment - or alternatively have not occurred within a required time-bound. Gauges may potentially operate according to an effective hierarchy where higher-level gauges take as input partial analyses from lower-level gauges.

Controllers encompass decision and coordination capabilities. Controllers decide whether certain conditions or incidents (detected by gauges) warrant system adaptation, and select the most suitable dynamic adaptation strategy amongst those known (or automatically inventible). Controllers instantiate and orchestrate effectors, which might include ordering or other dependencies among the effectors' tasks. Controllers may also reconfigure gauges and sensors (and themselves). While decision vs. coordination could, in principle, be separated in the architecture, the rationale for combining them is to simplify continual analysis during adaptation: the controller can then consider intermediate outputs from effector activities as well as gauges, possibly leading in some cases to rollback, retry and/or reconstruction of the adaptation plan, analogous to workflow exception handling.

Effectors are modules that tune the target system via exposed configuration parameters, perform partial replacement by initiating/retiring services, invoke special utilities such as the process migration, etc. Effectors are necessarily more tightly coupled to the target system than the rest of the architecture, often employing relatively *ad hoc* technology that can vary widely. Options include mobile agents, Java Management eXtensions (JMX) management beans, Simple Object Access Protocal-based (SOAP-based) interfaces, publish/subscribe events, etc.

Behavioral Models (see Figure 1) constitute an implicit component necessary to provide relevant information about the target system or its environment: what is its architecture and communication topology, how it is supposed to operate, what are its performance or security requirements or characteristics, and so on, possibly including negative models indicating expectations about what might "go wrong". These models are used to customize generic gauge and controller facilities to the specific target system, to indicate what to look for and what to do about it. The models do not necessarily have to be known *a priori*, but could be derived while the system runs. The set of models is intended to be open-ended, with new ones added or old ones updated or removed incrementally. Behavioral models need not exist as separate runtime entities, but would usually be deployed into the other components.

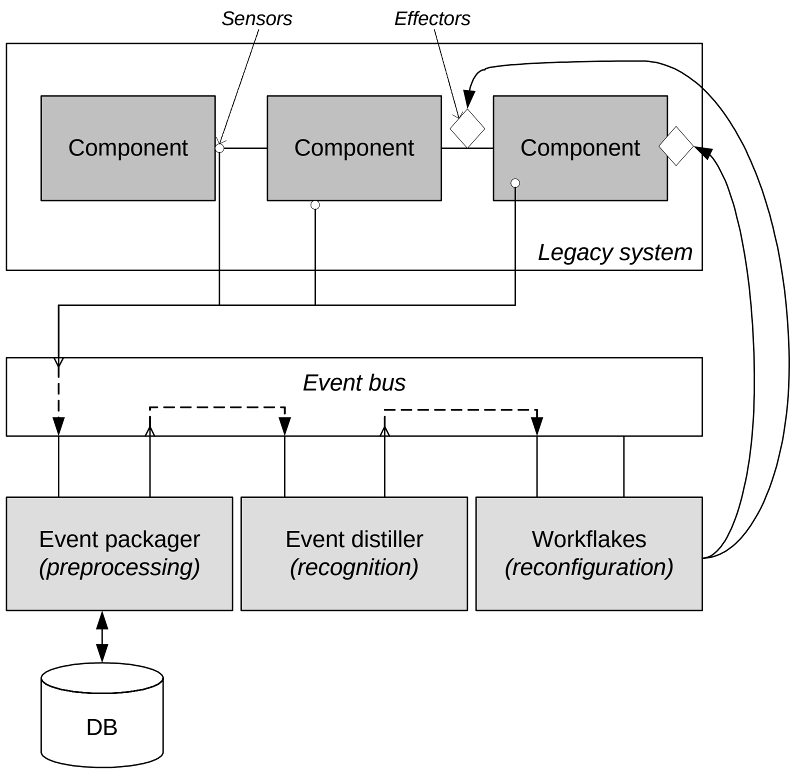


Figure 2. KX Architecture.

The final delivered version of the continual coordination and validation infrastructure is called Kinesthetics eXtreme (KX, pronounced "kicks"). As shown in Figure 2, communication among sensors, gauges and controllers was implemented via a publish/subscribe event notification system. Any of the then-available content-based messaging systems could have been used. The Siena event system from U. Colorado was chosen, as among the most advanced distributed event propagation systems where both source code and support were readily available. Some experiments were alternatively conducted with Elvin, originally from U. Queensland, Australia (since commercialized), due to its higher performance in cases where a single centralized router was acceptable. Communication between controllers and effectors was implemented point to point because of the need for synchronous interaction.

Sensors and effectors are selected as appropriate for the target system, and are not formally part of KX. In most experimental applications, Columbia's Worklet mobile agents (originally developed for the DARPA Evolutionary Design of Complex Systems (EDCS) program) were employed as effectors. The WPI part of the effort developed Active Interface Development Environment (AIDE) as one approach to sensor construction and deployment; that work is elaborated in the WPI volume of this final report. KX also includes an Event Packager (EP) component placed between the sensors and the event bus, to transform into the common event format, remove duplicates, timestamp events according to a globally synchronized clock, and act as a "flight recorder" to store both the raw events and their translations in a persistent log. This log supplies an SQL query interface for later “event mining”. EP also supports event replay, useful for debugging.

KX's gauge component, the Event Distiller (ED), performs temporal event pattern recognition across continuous event streams from multiple sensors. ED currently operates over events formatted as Siena's attribute-value pairs. ED is configured with manually specified “rules” defining the event patterns representing desirable and undesirable behaviors or conditions, basically in terms of states and transitions, with certain states performing success actions such as sending messages to the controller or triggering another state machine’s start state. ED’s rule language supports variable bindings, event reordering (within a time window), logic constructs, bounded and unbounded looping, chaining, absorption, etc. Incomplete state machines are garbage collected after time bounds expire, in that case invoking failure actions from matched states. A given KX instance consists of one ED component, which subscribes to all sensor events, and distributes those events internally among the newly initialized and partially matched state machines. Backtracking is not supported; instead multiple transitions may be followed simultaneously. ED does not always determine the “root cause” of a problem, but instead may detect that something “bad” has happened - or that something “good” that was supposed to happen did not.

KX’s controller is a decentralized workflow engine, Workflakes, which selects from a collection of predefined adaptation plans. The plan guides instantiation and launch of effectors to enact the workflow's individual tasks. Workflow tasks and plans have mostly been handcoded in stylized Java, although there was some experimentation with Columbia’s home-grown interpreter for the Little-JIL workflow notation from U. Massachusetts (UMass’ own interpreter was unavailable due to licensing restrictions). Workflakes is implemented via “plugins” into the blackboard-based task processors provided by the open-source Cougaar multi-agent platform developed by BBN and others. The motivation for employing workflow to close the feedback loop lies in the observation that adapting complex systems of systems often requires a multiplicity of fine-grained interventions impacting separate target elements. Those interventions may be conditional or otherwise dependent on others; during their enactment, certain effector tasks may fail, calling for contingency planning.

4.0 Detailed Task Descriptions

The originally proposed KX research and development effort consisted of five major sub-tasks: Active Interface Probing (AIP), Flexible eXtensible Markup Language (FleXML), Smart Event Active Connector Infrastructure (SEACI), Continual VAlidation COntrol Panel toolkit (CVACOP), and appropriate integration of all of these with relevant other DASADA prototypes.

4.1 Active Interface Probing (AIP)

WPI was primarily responsible for this subtask, which is reported in the companion WPI volume.

4.2 FleXML

Flexible XML (FleXML) is an extension of XML that loosens various restrictions of XML proper, to enable XML data from different vocabularies to be treated as if all part of the same “language”, for the purpose of cross-application, cross-domain, and cross-version filtering, aggregation and correlation. The approach is to delay binding of both syntax (schema) and semantics (tag processors) for XML fragments until needed. In particular, FleXML supports “cocktail” XML Schemas, where placeholders (processor instructions) are left to fill in new grammatical fragments on demand; smooth handling of data corresponding to older and newer versions of the same XML Schema; and introduction of special-purpose transformations and processing code for individual XML tags and attributes.

FleXML’s primary use in KX to date has been as an optional plugin for the Event Packager, to intelligently convert from XML-formatted sensor output (such as generated by AIDE) to the XML or non-XML event vocabulary expected by the Event Distiller - currently Siena events consisting of unordered sequences of typed attribute/value pairs. Siena supports a naïve translation from XML data into its flat event namespace, but cannot handle hierarchical formats.

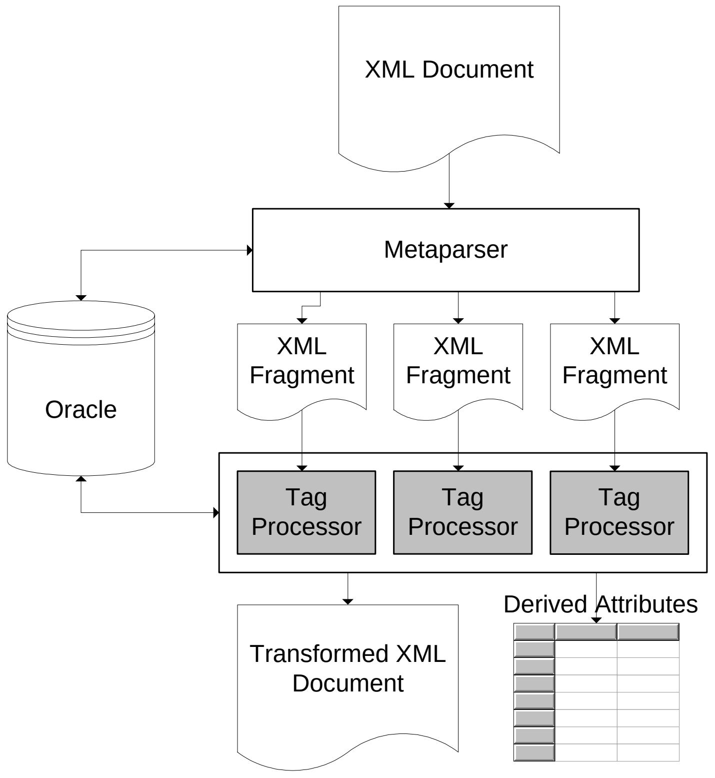


Figure 3. FleXML Internal Architecture.

4.2.1 The Metaparser

The *Metaparser* first performs a validating parse of a FleXML message (event). Like typical XML parsers, it performs *syntax* and *validity* checks on the XML document to make sure that it is well-formed and corresponds to a known schema. If either or both of these fail, a typical XML parser declares the document malformed or invalid, respectively, and stops parsing it. FleXML documents, like XML documents, must be well-formed. Rather than simply failing in the second error case (e.g., the next element tag is unknown or not acceptable at that point in the schema), however, the Metaparser attempts to resolve the problem autonomically. The apparently erroneous XML fragment, along with namespace and XPath context, is passed to a separate component (the Oracle) that contains a repository of schema information. If a matching subschema is found, it is installed into the Metaparser, which can then continue with its validation of the document. Note the problem fragment could appear "in the middle" of an otherwise valid message, so in effect the subschema is inserted into the parent schema at the relevant location.

Composition of multiple schemas within the same message, with dynamic handling of the subschema discovery, is desirable in several situations. Schemas may have been modified, the schema may be inherently "pluggable," as with the SOAP envelope schema, or the message format may have been designed as an elaborate composition of separate grammatical components (using the FleXML placeholders mentioned above).

As it parses each message fragment, the Metaparser calls the corresponding Tag Processors, if any happen to be associated with elements or attributes in the message. This is usually to rewrite or augment the original message, e.g., to standardize the format or highlight important attributes. Both tasks use the Oracle to allow adaptive, autonomic behavior in an environment of potentially changing message formats and their "meaning". Note this autonomic behavior is with respect to the KX infrastructure itself, as opposed to the target system.

4.2.2 Tag Processors

Tag Processors are XML Stylesheet Language Transformations (XSLT) or mobile code components. When the Metaparser hands off a message fragment to a Tag Processor, it also passes an "environment" associated with the message, where the Tag Processor can write its results as well as read results of previous Tag Processors. This allows state to be maintained as the message is processed by multiple components; the Metaparser and Tag Processors are stateless across messages, however, to reduce complexity and size, and improve performance. The Tag Processors, like the schema snippets, are discovered dynamically by querying the Oracle, allowing new Tag Processors to be deployed on the fly.

Tag Processors use a combination of XSLT patterns and XML-specified rules to write values to the environment and possibly rewrite portions of the message. The XSLT "rule template" can apply standard XSLT transforms, and/or add "virtual tags" to the message to identify particular side effects that should occur. An XML "rule set" describes the various possible effects, including writing of a particular attribute-value pair to the environment, and/or executing an arbitrary piece of code. Tag Processors are commonly used to standardize different message formats into a single format, to augment messages with higher-level information, and to run arbitrary legacy code modules against messages. The first two uses greatly reduce the burden on the Event Distiller.

4.2.3 The Oracle

The Oracle is a support database for the Metaparser and Tag Processors. When presented with an unfamiliar tag and the context in which it appeared (namespace and XPath), the Oracle will attempt to find and return an appropriate schema or schema fragment, along with associated tag processors.

4.3 Smart Event Active Connector Infrastructure (SEACI)

4.3.1 GC-4

Columbia originally proposed to develop GC-4, an improved (fourth) version of the Groupspace Controller event notification system it had developed for EDCS (culminating in version 2.5). However, the GC effort was temporarily abandoned in favor of using Siena, as recommended by Dr. Salasin. GC has been revived very recently under the new name MEET, for Multiply Extensible Event Transport, outside the scope of this effort.

4.3.2 XUES

XUES (XML-based Universal Event Service) consists of two main components, the Event Packager (EP) and the Event Distiller (ED). The “XML-based” part is a misnomer, since the selection of Siena as the event bus limited the use of XML in KX. At present, EP supports XML (via FleXML) but ED does not. Continuing work on ED, outside the scope of this effort, intends to add XML support as well as use MEET for event notification.

4.3.2.1 Event Packager

The Event Packager (EP) component is architected to support a number of event input services, such as duplicate removal, transformation, and persistent spooling. It utilizes a plug-in architecture to support a broad variety of incoming event formats (*inputs*); a variety of transformations, including persistent spooling and timestamping; and a variety of output event formats and other options (*outputs*). New plugins can easily be synthesized; for example, Instant Messaging (IM) messages can be represented as a form of event input.

The various plugins are coordinated via a user-definable metabase (metadata database) that dictates what should be done to the data (*transforms)* and where the data should be sent. Transforms can include single-event processing tasks, such as event clock/timestamp synchronization, static event reformatting/rewriting, augmentation, and selective or complete event persistence. Typically, the goal is to have a number of different input formats streamlined, spooled, and aggregated onto one event stream for the other KX components.

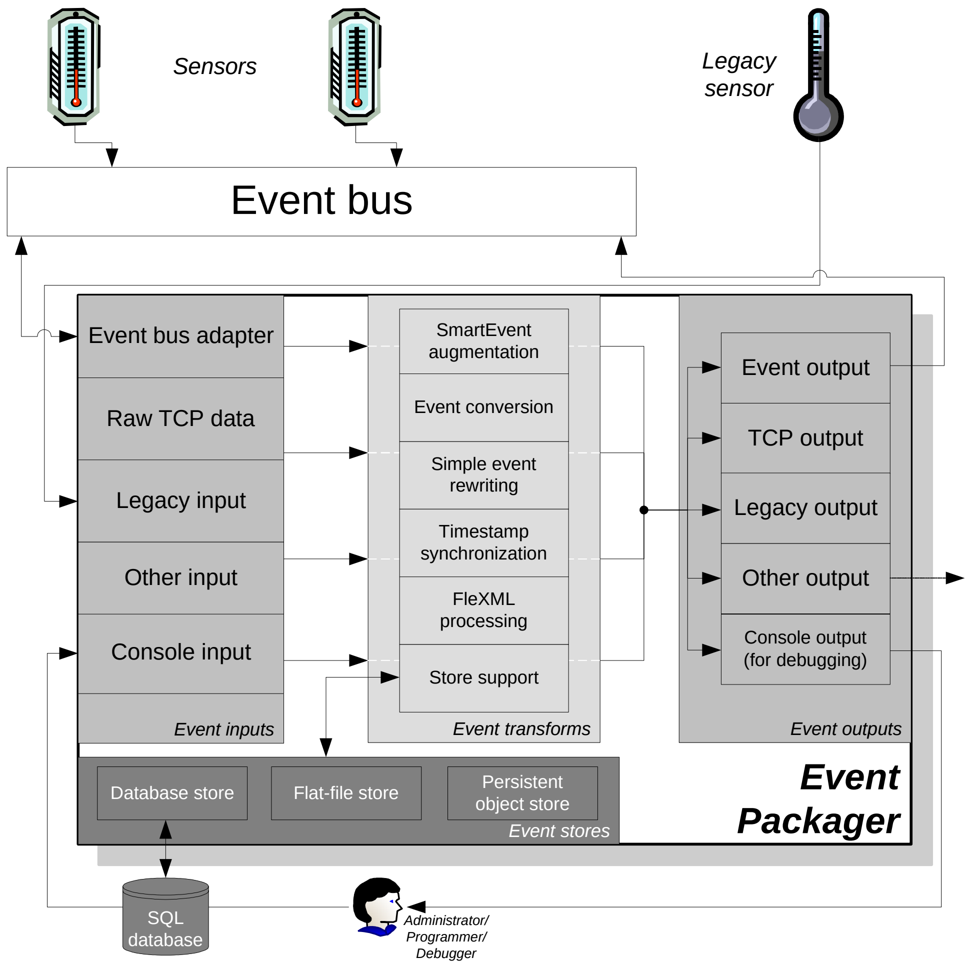


Figure 4. Event Packager Internal Architecture

The Event Packager implementation was designed from the ground up to be easy to extend. Developing a new input, output, transform or store only requires that one Java class be extended, and some simple methods filled in. This enables the quick and easy creation of wrappers around existing sensors and middleware. The Event Packager uses its own opaque event format container to allow future support for new event formats without breaking compatibility with existing plugins (although for optimal performance, certain plugins might support introspection into event formats for specialized processing). By using opaque event containers, minimal per-event decision-making is needed, which enables the creation of fast pseudo-pipelined datapaths. If more complex processing is needed, a transform can be applied - although this may affect event processing speed.

Columbia has developed input plugins that support and standardize Siena and Elvin messaging, TCP socket streams (transporting both serialized Java objects and XML messages), console input, email (via sendmail), and AOL Instant Messaging (AIM) messaging as an event source. A broad array of output formats closely mirrors these inputs. Transforms include event conversion (from Siena and ASCII formats) and event timestamp synchronization (to compensate for distributed clock environments).

EP also supports in-memory, JDBC-backed SQL, and flat-file (serialized object) stores. Persistent logging enables the Event Packager to support "latecomer" analysis, or "reanalysis" of previously-received event streams, as new Event Distiller gauges are deployed. Multiple persistence techniques may be simultaneously employed via the plug-in model, so that rules can specify persistence to one or more data repositories, such as an SQL database, which enables the use of efficient offline analysis and data mining.

The above components are arranged on-the-fly. Upon startup, the Event Packager reads its configuration file and instantiates the necessary plugins and begins routing events. However, plugins can be added (via Java late-binding reflection mechanisms) and removed during runtime.

EP currently consists of about 9,000 lines of Java code; the core engine that coordinates inputs, transforms, outputs and stores is about 2,000 lines, while the bundled plugins to deal with input, output, transform and store comprise the rest. Some C glue code handles legacy integration. The optional FleXML facility adds 2,200 lines of C++ and Java plus some XML files for the Metaparser and Oracle, and about 4,500 lines of Java for the accompanying Tag Processor architecture. A typical rulebase is usually a few hundred lines of XML.

4.3.2.2 Event Distiller

The Event Distiller (ED) is KX's main gauge component. It performs sophisticated, possibly cross-stream temporal event pattern analysis and correlation across continuous data streams from multiple sensors, to monitor desirable and undesirable behaviors. When undesirable behaviors occur (or desired behaviors do not occur within the requisite time-bound), the Event Distiller generates meta-level events indicating this interpretation; these higher-level notification events also carry information about the lower-level sensor data that contributed to the analysis. (The original proposal included a separate "Event Notifier" component to generate these meta-events, but the implementation subsumes this function into the Event Distiller.) The Event Distiller is dynamically configured with the "rules" defining complex event patterns of interest – that is, the behavioral models regarding what to monitor – so new models can be added and previous models replaced or removed on the fly.

Thus ED is responsible for detecting problematic or anomalous system activities by matching (*gauging*) sequences of events emitted by one or more sensors. An *event sequence* is defined here as being a nondeterministic ordering of events ultimately indicating correct vs. incorrect behavior. Such an event sequence's transitions (between subsequent events) will almost always have timebounds so as to emphasize the real-time nature of the application domain and to act as a check on the nondeterminism.

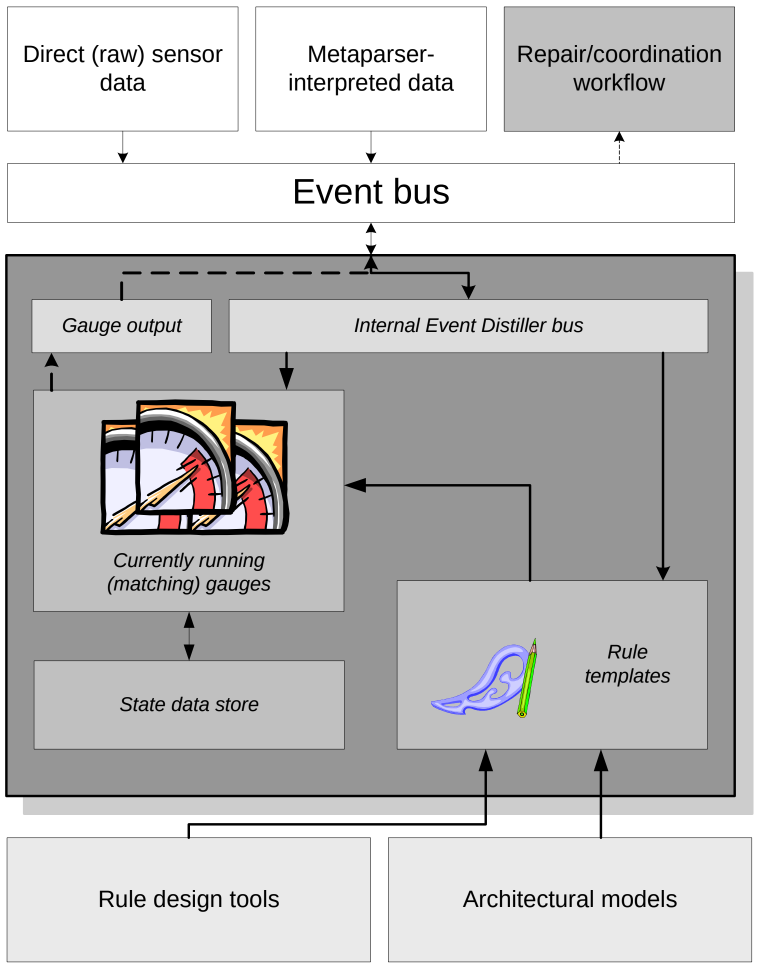

Direct (raw) sensor data
Metaparser-interpreted data
Repair/coordination workflow
Event bus
Gauge output
Internal Event Distiller bus
Currently running (matching) gauges
Rule templates
State data store
Rule design tools
Architectural models

Figure 5. Event Distiller Internal Architecture

An ED rule is partitioned into "states" and "actions", where matches amongst the former are mapped to (meta-)events that are emitted that correspond to the latter. This state/transition representation closely corresponds to a nondeterministic finite automaton – the idea is that one event may lead to many different possible subsidiary events, and one wants to match whichever ones are appropriate. Transitions are inherently timebound to provide a control on the size of the nondeterministic matching problem – an expiration implies that a transition is no longer possible, and implementations can then garbage collect that test from the pool of potential matches for incoming events to reduce the amount of system state required during execution. An alternative approach would be to provide backtracking, but this is impractical given the runtime requirements of such a system and the potentially huge number of events it may witness at any given time.

The ED internal architecture supports several additional first-level constructs as defined in the rule language:

- Rule chaining is accomplished by allowing published actions from one rule to match other rules' states. This late-binding approach enables dynamic rules to be created and to immediately support chaining.
- Looping provides Kleene star-like functionality, but can also match a specific number of times.
- Success and failure actions can be made at any matched state. A success action is published immediately upon reaching the state. A failure action is one where all the transitions from that state to another state are eliminated and no further transitions can be done, and is sent upon successful garbage collection of the current rule instance. Multiple success and failure states can be specified at each state if desired. Such actions may trigger a rule chaining within the Event Distiller architecture, may be used by other interested components (such as controllers that begin applying a repair or reconfiguration workflow), or may even trigger human notification via some immediate communication channel, such as a pager.
- Absorption enables a given state match to be exclusive, e.g., if a particular state of a particular rule enables absorption, all rules below it will *not* match state, even if they specify the exact same criteria as the first matched rule. Note that this implies an ordering upon the rulebase, e.g., rules at the top have absorption capability over all other rules in the rulebase, whereas rules at the bottom can declare absorption but such a declaration has no effect.
- Variable binding enables conditional matching – a value can be *bound* by the first match, and further states in a rule may require that value to appear in subsequent events. This is useful for any sequence of events that refer to a common shared value, such as the name or unique identifier of a service being monitored.

Internally, the Event Distiller uses a collection of nondeterministic state engines for temporal complex event pattern matching. The rulebase is loaded into memory, and forms a series of "state machine templates"; once an event matches the *first* state of one of these templates, an instance of the template is automatically created to keep progress of the matching through the state machine. While this is memory-intensive, it allows a richer representation of event sequences: logic constructs are supported, as are loops, rule chaining, and variable binding as required by the architecture. Memory usage is mitigated by supporting timeouts and automatic garbage collection. Timestamped event reordering is also supported, so if events arrive out-of-order within a certain window (1 second by default), the Event Distiller will rearrange them appropriately so that sequences, and causality, can still be recognized correctly. Note that such reordering, if done with many sources, requires some authoritative time declaration as close as possible to the sources themselves, as network latencies may be unpredictable. If the generator of the events being matched doesn't support timestamping, Event Packagers may be placed at the generation point or at its immediate network peer to create timestamps to enable reordering.

The Event Distiller's repertoire of event patterns may be populated in one of several ways: First, an XML-format rulebase is supported, where event sequence patterns are specified, along with timebound parameters among sequence elements as well as “success” and “failure” notifications. There is also a GUI to assist a KX integrator (known as Continual VAlidation COntrol Panel Toolkit (CVACOP) in the original proposal); it also works as a systems management console for human engineers, although a major goal of the effort has been to automate many repairs within a KX feedback loop (via notifications to Workflakes). Second, the Event Distiller supports dynamic rule generation – messages can be sent to the Event Distiller with XML snippets specifying a rule or a segment of a rule (e.g., to construct new rules on the fly or modify existing rules). Currently, such rule modifications are received through the publish-subscribe channel, potentially containing an XML snippet that contains a full rule (e.g., states and actions) be matched from hereon. Such rule changes affect *templates* and not currently-matching rules (i.e., for future matches). Third, as with the Event Packager, other sources can be easily integrated: For instance, support for CMU’s Acme architectural description language constraints has been partially integrated: The Event Distiller can act as a “reporting gauge” onto the Acme Gauge Bus, thereby providing feedback to the corresponding architecturally-oriented repair tools.

The Event Distiller implementation is in Java, currently about 7,000 lines of code. The event pattern rulebase may vary in length depending on the complexity of the behavioral model, but in the experiments were typically a few hundred lines of XML. Note that an Event Distiller rulebase defines the event patterns of interest, from the behavioral models, and is not related to an Event Packager rulebase specifying plugin configuration.

4.3.3 Gaugents

The Gaugents concept presented in the original proposal was based in part on the Worklets mobile agents first investigated for DARPA’s EDCS program, and in the end the Worklets name was retained. However, Worklets were completely redesigned and reimplemented entirely in Java (about 4,000 lines of code), with none of the original JPython system reused. Worklets have been employed primarily as KX effectors, for direct manipulation of the target system being dynamically adapted, but have also been tried as a sensor technology, in that guise called “probelets”.

Worklets carry self-contained mobile code that can act upon target components and follow directives indicating their route and operation parameters. Worklets provide a means for flexible software (re)configuration, with effects local to the target component. Each Worklet works with the component needing (re)configuration, deciding what to do on the basis of the component state and its own capabilities. Moreover, a Worklet can “hop” from one component to another, carrying out at each location a portion of a predetermined multi-step configuration sequence. A very simple example of this kind of reconfiguration would be dispatching a Worklet to modify the ports used for inter-communication by two components of the target system: the Worklet would carry information about the various port numbers and code to activate new ports and deactivate old ones.

Each Worklet can contain one or more mobile code snippets (called *worklet junctions*) that are suitable for actuating the required adaptation of the target system. Junctions’ data structures can be initialized with data, typically coming from the repair task definition, its encompassing repair process context, and the information contained in the event(s) that represents the triggering condition. Furthermore, any process-related configuration of Worklets is accounted for by *worklet jackets*, which allow scripting of certain aspects of Worklet behavior in the course of its route. Among them, preconditions and timing for junction delivery and activation, repetition of junction execution, exit conditions for the junction's work, directives to supersede, suspend, and reactivate a junction upon delivery of another one, and so on.

The separate adaptation controller (e.g., Workflakes) requests junctions for the dynamic adaptation task at hand from a Worklet Factory, which has access to a categorized semantic catalogue of junction classes and instantiates them on its behalf. Interfaces exposed by junctions in the catalogue must be matched to the kind of capabilities that are necessary for the task and to descriptions of the target components subject to dynamic adaptation.

Once a Worklet gets to a target component, the interaction between the junction(s) it carries and that component is mediated by a *host adaptor*, which semantically resolves any impedance mismatch between the interface of a junction and that of the component (see Figure 6). The original purpose of the host adaptor was to provide each worklet junction with a consistent abstract interface to a variety of component types, including COTS or legacy components, that can be subjected to forms of dynamic adaptation that are *semantically* equivalent from the worklet junction's perspective.

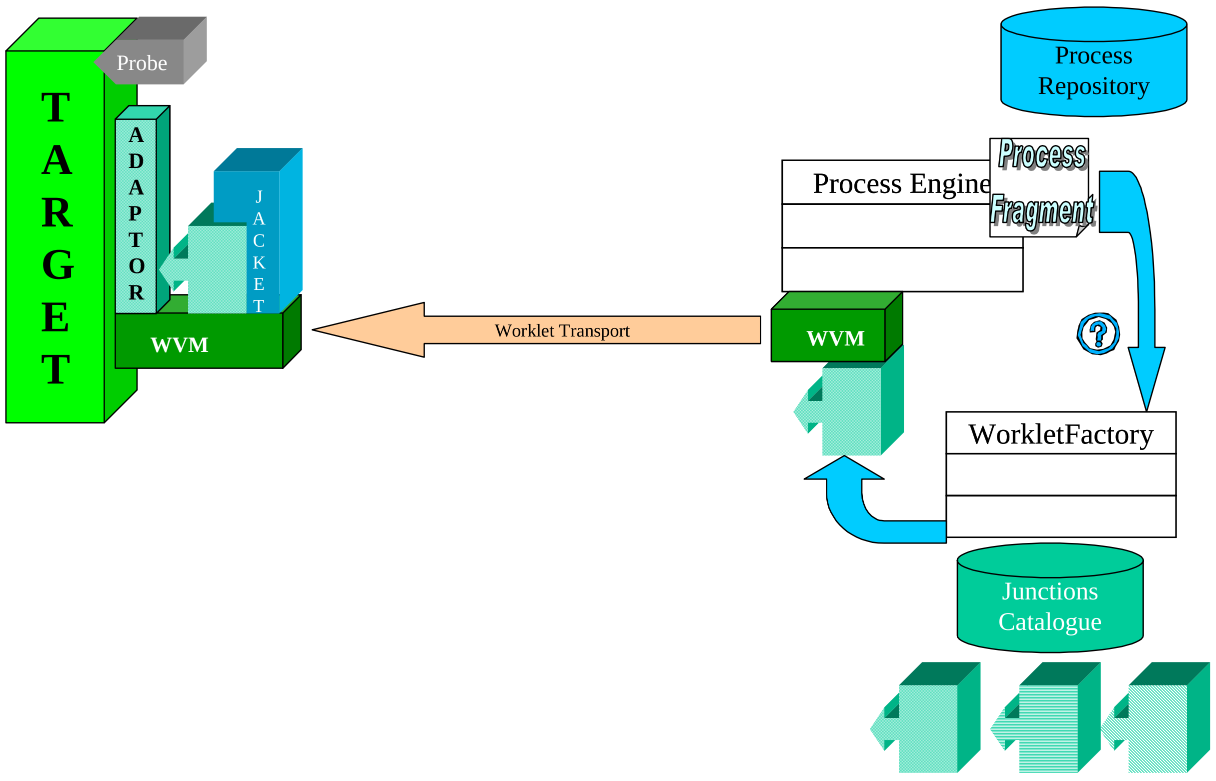


Figure 6. Selecting Junctions and Shipping Worklets

There is a tradeoff, however, between the complexity of the host adaptor and the complexity of the worklet junctions it supports. The first realization of Worklets used JPython scripts for the entire worklet body, which allowed very simple Worklets but required a very heavyweight host adaptor (including, among other facilities, the entire JPython interpreter) – which made it unsuitable for the constrained-resource devices (e.g., PalmOS or PocketPC PDAs). Requiring the identical abstraction on such devices as on, say, conventional website servers, was prohibitive. The current implementation instead uses Java code augmented by Columbia's own peculiar jacket scripting, which allows for lightweight/subset JVM implementations and relaxed requirements on host adaptors, e.g., in the simplest case to directly expose the local API "as is" - but of course requires more specialized knowledge and advanced capabilities on the part of individual worklet junctions.

The transport services, as well as the worklet junction execution context and the interpretation of jacket scripts, are provided by a generic *Worklet Virtual Machine* (WVM) located at each target component intended to host Worklets (generally integrated into the host adaptor). For each dynamic adaptation task, the controller typically schedules the shipping of multiple Worklets. Each Worklet may traverse various hosts in its route, installing junctions in the corresponding WVMs. Execution of the procedural code of the junctions is (optionally) governed by jackets and carries out adaptation of the target component through the programmatic interface provided by the adaptor. Junctions' data structures may be modified as a result of those operations. At the end of its route, the Worklet may go back to its origin, for any reporting and housekeeping needs, which are performed by a specialized origin junction.

4.3.4 Continual VAlidation COntrol Panel toolkit (CVACOP)

CVACOP (later renamed to TRIKX, Transitional Interface for KX) turned out to be a very minor aspect of this work. It is discussed briefly as part of the Event Distiller discussion above.

4.3.5 Integration with other DASADA contractors

Most of the integration with prototypes developed by other DASADA participants is discussed as *a propos* above and below. Software and/or notations from U. Colorado, BBN, CMU and U. Massachusetts were utilized most extensively. We also collaborated closely with Teknowledge and OBJS (as well as BBN and CMU) on the standard reference architecture depicted in Section 1 and on standard APIs for sensors and gauges, and with ISI on applications (notably GeoWorlds).

4.3.6 Other - Workflakes

The original proposal did not include what became one of the most significant results of the effort: the Workflakes controller, which enabled closing the feedback loop – that is, autonomically performing continual coordination as well as continual validation.

To automate responses to the frontend monitoring, the output of gauges is input to a backend decision process that determines whether/how the target system must be adapted. In the simplest case, some gauge may assert a fact that already carries with it unequivocally defined consequences. Other times, a variety of tools could be exploited for decision support: for example, formal architectural knowledge models of the target system, coupled with constraint analysis and architecture transformation tools (e.g., CMU's AcmeStudio).

When a decision to apply some adaptation is made, a single action will sometimes suffice to fulfill it. In most cases, however, the adaptation will have to be mapped onto several finer-grained and concerted activities, impacting several implementation-level elements. Then a sophisticated coordination mechanism is needed: some of those activities may be conditional, or dependent on others, or may fail, calling for contingency planning, etc.

To address that complexity, Workflakes relies on process-based coordination, and treats gauge outputs as input triggers for the enactment of a tailored adaptation process. The choice of process technology as the coordination paradigm for dynamic adaptation is motivated by its ability of expressing even very complex patterns of coordination and dynamic dependencies in an explicit and abstract way that is also executable and reusable. Furthermore, process enactment engines are increasingly mature, even when applied, as in agent-based systems and Enterprise Application Integration (EAI), to completely automated rather than human coordination subjects.

A Workflakes process unfolds according to a task decomposition strategy, which in the end generates, configures, activates groups of effectors, and coordinates them towards actuating the desired side effects onto the running target system. Effectors are considered a first-class resource: they must be explicitly described in the process enacted by the Workflakes engine.

The impact of effectors can range from the adjustment of a single operation parameter, to a method call, to complex reconfigurations of the target architecture - involving many components and connectors at once. Similarly, the technologies that can be used to implement effectors may greatly vary, depending on their reach as well as the nature of their target: they are often the most target-dependent elements in our approach, and are likely to be handcrafted. However, a significant amount of standardization of the interface between the process engine and the effectors it coordinates can be achieved, and can help decoupling the control and the actuation layers across technology and application contexts.

Such an interface is relatively simple at the conceptual level, and requires means for the process engine to *look up*, *instantiate* or *recruit*, *configure* and *activate* the effectors, and for the effectors to *report back* the outcome of their work in a meaningful form for the process engine. The implementation of this conceptual interface of course varies depending on the technological underpinnings of the effectors employed.

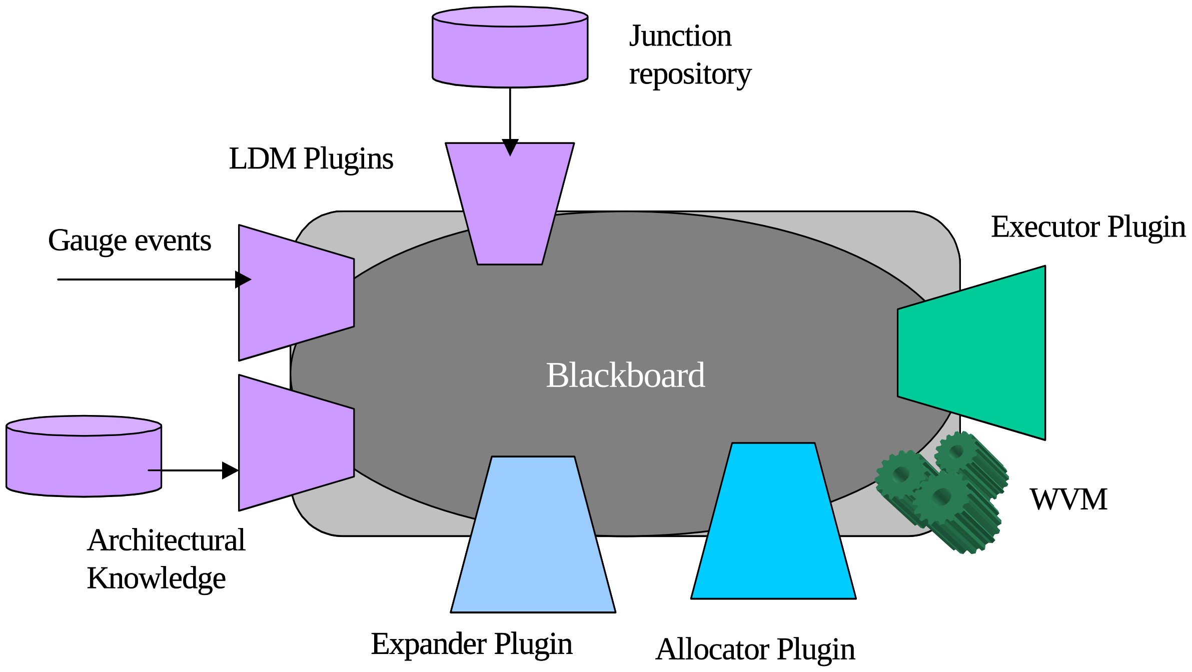


Figure 7. Representation of a typical Workflakes Task Processor

The current implementation of the Workflakes runtime engine relies on a specialization of the Cougaar open–source distributed platform. Cougaar's decentralized task processors provide us with a number of largely autonomous controllers for the enactment of distributed dynamic adaptation processes. Each task processor is further specified as a set of Cougaar plugins. Plugins allow customizing the functionality of task processors by inserting components that implement a particular logic or specific capability. As shown in Figure 7, a typical Workflakes task processor includes several of what Cougaar calls Logic Data Model (LDM) plugins, which are used to import and convert KX gauge-generated meta-events in terms of process facts, maintain internal knowledge about the target system and its state, and access a repository of effectors; an Expander plugin to load process definitions and spell them out as hierarchical decompositions of tasks; an Allocator plugin to map tasks to effectors and target components as needed; and an Executor plugin that handles the instantiation and shipment of effectors.

Workflakes task processors interact with the Worklets mobile agents utility in two different ways. Some Worklets implement effectors that carry out the details of the dynamic adaptation process onto the target system components and connectors. One of the major responsibilities of the Workflakes process is therefore to decide what Worklet mobile code needs to be dispatched for a given dynamic adaptation task. That is why the repository of effectors descriptions (see Figure 6) is an essential component, and the corresponding effectors are treated as first-class process resources.

Workflakes also uses Worklets to load process definitions on the fly onto task processors, either with a pull or a push modality. The rationale for such a facility is that the dynamic adaptation of target applications is likely to call for dynamically adaptable controllers and coordinators. In Workflakes, the plugins of any task processor are initially idle and devoid of any hardcoded logic related to any particular process; for that reason, they are called *shell plugins*. The set of shell plugins launched within a task processor at start time is therefore merely indicative of the kinds of service and functionality that the cluster is meant to offer within the overall distributed Workflakes engine. Shell plugins can be activated at any time via the injection of specific process definition Worklets. Those Worklets dynamically deploy process fragments to the most convenient task processor for execution. Only after such deployment, shell plugins acquire a definite behavior, and start taking part in the enactment of the process.

Such a process delivery mechanism is effective for a centralized as well as a more scalable, decentralized process enactment architecture. It may, for example, be used in the pull modality to incrementally retrieve process fragments from a process repository when requested to handle certain specific adaptations, or in the push modality for on-the-fly process evolution across a distributed Workflakes installation.

Workflakes, not including Cougaar or Worklets, consists of about 2,200 lines of Java code.

5.0 Experimental Results

Several experimental KX applications are described, demonstrating the wide applicability of the approach.

5.1 Service Failures

KX was integrated with a complex GIS (Geographical Information System) intelligence analysis tool developed at the USC Information Sciences Institute (ISI), known as GeoWorlds. GeoWorlds uses a distributed set of services glued together by Jini. While the system generally works well, sometimes services stop running, with no recourse except to wait for the request to time out and manually restart the appropriate backend subsystem. For example, its reliance on harvesting standard websites (e.g., www.bbc.co.uk), for news items that maps to locations in GIS, is subject to frequent glitches.

```
<state name="Start" timebound="-1" children="End" actions="" fail_actions="">
  <attribute name="Service" value="*service"/>
  <attribute name="Status" value="Started"/>
  <attribute name="ipAddr" value="*ipaddr"/>
  <attribute name="ipPort" value="*ipport"/>
  <attribute name="time" value="*time"/>
</state>
<state name="End" timebound="15000" children="" actions="Debug" fail_actions="Crash">
  <attribute name="Service" value="*service"/>
  <attribute name="State" value="FINISHED_STATE"/>
  <attribute name="ipAddr" value="*ipaddr"/>
  <attribute name="ipPort" value="*ipport"/>
  <attribute name="time" value="*time2"/>
</state>
```

Figure 8. Failure Detection Pattern

WPI's AIDE was used to automatically instrument the GeoWorlds Java source code, and in particular the mechanism that dealt with request-to-service dispatch, with sensors that monitored the start and end of method calls relevant to contacting external services. The Event Distiller incorporated rules to monitor a variety of method calls, making sure that a "termination" call matched up with each "initiation" call within an appropriate timebound (ranging from seconds to a minute).

Figure 8 shows a simple event pattern used to perform failure detection. The incoming sensors reporting Status and State values track method completion. If for some reason a "FINISHED_STATE" was not received within 15 seconds after a method had initiated, the system sent out the "Crash" event; otherwise, the (misnomer) "Debug" notification would be emitted, signifying a "success". Note that the strings prefixed with an asterisk ("*") designate a variable binding, e.g., the Event Distiller substitutes all instances of "*service" by the first source that it sees for this instance of the rule. Thus this one rule can match a large number of different sources and subjects.

When Workflakes received a "Crash" event, the repair involved a simple restart of the service specified in the message generated by the Distiller. A more sophisticated repair, which was not implemented, would have coordinated multiple services to prevent having to restart a long transaction from scratch, instead using partial results preceding the individual service failure.

5.2 Load Balancing

Several GeoWorlds execution scripts relied on computationally-intensive backend services, such as a noun phraser that analyzed incoming news articles and extracted nouns for mapping to GIS attributes; crash avoidance and performance maximization through request relocation was clearly desirable. To accomplish this, the relocatability of Jini services was exploited to build a load-balancing solution for GeoWorlds. A system monitor sensor measured the overall load on the backend system(s) running the noun phraser.

CMU's Acme architectural description language was used to specify constraints. The Acme Gauge Extractor constructed Event Distiller rules based on the predefined architectural constraints. The Event Distiller then acted as a "reporting gauge" onto the Acme Gauge Bus, thereby providing feedback to the corresponding architecturally-oriented repair tools. An Acme architectural description of GeoWorlds included specified system load constraints on the appropriate services. During the execution of various services, if this load exceeded a predetermined threshold for an extended period of time (as specified by the Acme constraint), the Event Distiller detected and reported it as a violation of the architectural constraints. The triggered repair caused the service to move to a different Jini-enabled host. The load and service state were also visualized using AcmeStudio's architectural diagram tools, so one could watch the feedback loop in action.

Additional logic was programmed into the Event Distiller rulebase to detect oscillation. In particular, if many Event Distiller messages requesting a load-balance were detected within a short timespan, one of two tacks could be selected: either eliminating load balancing between the two oscillating hosts for future repair plans, or increasing the load threshhold in the architectural constraints.

5.3 Quality of Service

Columbia had the opportunity to experiment with a commercial J2EE-based multi-channel Instant Messaging (IM) service used daily by thousands of real-world end-users. First, on-demand scalability was added: by probing user sign-on events and server request queues, KX was able to determine the load of each member of the IM server farm and take appropriate actions whenever needed. Repairs, selected on the basis of the inferences carried out using Event Distiller rules, encompassed modifications to the threading model of active servers, or even on-the-fly deployment and activation of additional server instances and corresponding reconfiguration of the commercial load-balancer of the IM server farm to redirect client traffic to these new servers. Failure detection was also supported from a load-balancing standpoint: information on server failures and interconnections between servers and backend DBMS entities was similarly captured to facilitate load balancer reconfiguration to direct client traffic to still-functional servers. The same set of sensors and effectors, coupled with slightly different Event Distiller gauge rules and Workflakes repairs, were also used to support controlled and graceful staging of the service infrastructure; this enabled automated software release deployment without necessitating a complete shutdown (and service interruption) during transitions.

A set of quantitative results were derived from running and observing the adapted IM system in lab conditions, with both manual and automated traffic simulation that reproduced in-the-field demands on the IM service. These results focus on the improvement via automation in the support, maintenance and management activities typically carried out on the IM service under field conditions. Also, some measurements about the development and integration effort necessary to implement the case study were taken. The most significant quantitative results are:

- Substantial reduction in effort for deployment and configuration of an IM service in the field, originally around ½ person-day, with locally present experts. With KX, that was reduced to 1-2 minutes from a remote location.

- Reduced monitoring and maintenance effort necessary to ensure the health of the running service. KX completely automates the 24/7/365 monitoring of a set of major service parameters, as well as the counter-measures to be taken for a set of well-known critical conditions.

- Reduced reaction times and improved reliability: for example, KX recognizes the overload of an IM server in 1-2 seconds and takes approximately 40 seconds to put in place an additional server replica. Overload detection was originally manual, starting with accumulated application logs – a clearly error-prone approach, potentially endangering service availability.

- Manageable coding complexity: KX sensors, gauges and effectors were derived from generic code instrumentation templates and then customized with situational logic. This results in rather compact code: 15 lines of Java code on average for sensors, and usually less than 100 lines for effectors. The total code written for this specific experiment on top of the generic monitoring and dynamic adaptation facilities provided by the KX infrastructure was approximately 2,000 lines of Java and XML code.

This study demonstrated the utility of a KX end-to-end feedback loop for service management and application-level QoS in an industrial context. Traditional application management practices report warnings, alarms and other information to some knowledgeable human operator who can recognize situations as they occur and take actions as needed – with very limited automation in the management platform. Instead, the KX approach offers a high level of guidance, coordination and automation to enforce what is a complex but often repeatable and codifiable process.

5.4 Spam Detection and Blocking

In order to demonstrate KX's flexibility beyond the more conventional system management cases above, Sendmail, a popular email Message Transfer Agent (MTA), was instrumented to capture messages being received in a target network. A *Sendmail milter* was installed to capture incoming traffic. The Event Distiller rules (see Figure 9) would trigger if multiple (3+) messages containing the same source and Message-ID were received in a very short timespan (under 10 seconds). Once detection has occurred, a mobile agent effector is dispatched to reconfigure the Sendmail MTA to block all further messages from that source address, by rewriting the configuration file and sending a hangup signal (SIGHUP) to Sendmail to reload its configuration.

This solution worked for simple spam – i.e., one message sent by a spammer to a broad number of people in the same organization would verifiably get caught and future communication from that spammer would be blocked. While this technique is superceded by better spam-specific technologies, such as SpamAssassin, which uses dynamic rules and Bayesian learning to distinguish more "stealthy" spam, this example demonstrates the broad utility of our Event Distiller's timebound-based pattern matching, in this case with email-specific semantics.

```
<state name="a" timebound="-1" children="b">
  <attribute name="from" value="*1"/>
  <attribute name="messageID" value="*2"/>
</state>
<state name="b" timebound="100" count="1" children="" actions="A,B" fail_actions="F" absorb="true">
  <attribute name="from" value="*1"/>
  <attribute name="messageID" value="*2"/>
</state>
```

Figure 9. Sample Pattern to Detect Repeated Emails

5.5 Multimedia Synchronization across Multiple Users

Adaptive Internet Interactive Team Video (AI2TV) is an NSF-funded project concerned with, among other things, synchronization of "distance learning" lecture videos among dispersed students studying together or working on a team project. The students are assumed to watch from their homes, with dialup, DSL or cable connections. Another faculty member at Columbia developed techniques whereby the mpeg video is "semantically compressed", in the lower bandwidth cases as sequences of jpeg images automatically selected as the "best" rendition of the video given the bandwidth available (as opposed to, e.g., sending every Nth frame). These sequences are precomputed for several anticipated bandwidth levels.

The DASADA PI's part of this non-DASADA effort is to synchronize among the student viewers such that they all see "the same thing at the same time" in the semantic sense, even though it may not be the same video frame, minimizing skew. It was found that synchronizing at the video server in a push model does not work very well, since it does not take into consideration what is actually viewed at the client.

This problem provided an unexpected opportunity to employ the KX technologies developed for DASADA. The approach taken was to add a separate feedback control loop that monitors the video clients and can dynamically adjust their configurations and parameters while they continue operation. Sensors were inserted into the clients to determine what frames are actually showing, e.g., at 3 second intervals, and what is the actual current bandwidth. The sensor data from all clients is input to the Workflakes controller, which then instantiates and coordinates local effectors that dynamically adjust for each client the selection of which compression level and which next frame to pull from the video server. This computation considers what is already in the client's cache, since Workflakes also prefetches from possibly higher resolution (lower compression) streams into client caches during idle time, e.g., when the users have “paused” the video to discuss a point.

Experimental trials have used lecture videos taped by the Columbia Video Network (http://www.cvn.columbia.edu). Bandwidth was artificially throttled in the lab. An invented “goodness” metric, which weights aggregate resolution and skew, showed very nice results to date. Measurements are continuing with alternative weighting schemes, to distinguish the contributions of synchronization vs. prefetching.

Notice this problem could not possibly be addressed by conventional human systems management approaches, due to the time scale of the synchronization. The successful application of KX components in this case demonstrates the potential to address continual coordination and continual validation of real-time software systems.

6.0 Limitations and Suggestions for Future Work

6.1 Gauge Issues

The Event Distiller (ED) should be significantly redesigned and extended. The hypothetical new ED2 would still recognize complex temporal event patterns, as at present akin to Stanford's earlier work on Rapide, but in the future would support enhanced timing-related operators and variable bindings, e.g., to enable detection of longer-term trends.

One significant goal for such an ED2 would be to perform general correlation across events from two or more sensor streams, and notably to support longitudinal correlation over extended periods of time. For instance, consider the case of correlating alert streams from several different network-based surveillance detectors and/or host-based anomaly detectors in the security domain, to attempt to reduce false positives and pinpoint escalating incidents. It would be desirable to make these thresholds dynamically adjustable, understanding that increases in the number of prospective matches contributing to a final state as well as the length of time to wait for that final state will likely increase runtime overhead and decrease performance for other matches. Then one could comparatively exploit the Event Packager’s event mining interface (never used to date) for long-term pattern durations and/or many-stream correlations, which would not permit near-real-time recognition (and thus delay system management response).

It would also be desirable to investigate fast matching algorithms considering very large numbers of sensor event inputs, in particular germane for (but not limited to) the multi-stream correlation issue. One idea is to reconsider the old RETE, TREAT and Gator discrimination network concepts used in AI production systems and databases. Although this typically memory-intensive approach has arguably been beaten to death, some more recent rule-based systems have extended to time-stamped events and temporal constraints between events. One possibility is combining an open-source implementation of RETE, called DROOLS (http://drools.org/), into a blackboard architecture, as a mechanism for sharing persistent subexpression matching among temporal state machines within an ED2 instance.

A complementary approach might be to recognize patterns over *vectors* of events, where each vector element corresponds to the latest value from a given sensor stream - which may ease handling of patterns across sources that emit events at greatly different time intervals. This idea is inspired by Cornell's Astrolabe system, which maintains an analogous vector, which is not used in event pattern recognition directly, but instead provides a distributed repository for use by separate monitoring applications. An extension might be a "sliding vector window", essentially a matrix, where the front row is the vector of latest values as above, and the remaining rows are past values. These might be the last N values, or the values at N previous time epochs uniform across the vector (which might be different from the first case if the sensors emit at different rates), or the N most "interesting" values in time order, where "interesting" could be determined differently for each vector element (sensor), e.g., different from the previous recorded value by more than some threshold, or over/under some absolute threshold(s).

Another topic not yet investigated would be to deploy multiple instances of ED2 in the same KX instance, usually but not necessarily with different rules, without unnecessary duplication of effort. As a first step this would mean any given ED2 instance should only subscribe to events that could plausibly match its current pattern base, so it must automatically convert from its rules' state transitions into subscriptions for the content-based publish/subscribe event system. Since the rulebase can be dynamically updated, adding and deleting rules, it would be necessary to develop algorithms for efficient incremental subscription update – a general concern for pub/sub systems since subscription churn can prove very expensive. Most systems assume a relatively low churn rate, e.g., recent Siena experiments assumed subscription changes on average once every ten minutes, which may not be adequate for many DASADA-like applications.

One purpose for deploying multiple gauge engines would be simply locality, to better handle multi-enclave target applications where most patterns of interest occur within a LAN, but some (e.g., management of multi-mirrored server farms) involve wide-area distribution where appropriate gauge (or gauge element within a hierarchy) placement is an issue. There may not be a single placement algorithm that takes all factors into account; so different algorithms and heuristics should be investigated. Another likely purpose for multiple gauges is load balancing, which should be considered in tandem with handling very large numbers of sensors.

Another interesting case that would take advantage of a hierarchy (or lattice) among gauge units is to represent short-term problems and apparent non-problems as meta-events subscribed to by specialized ED2 instances analyzing longitudinal trends, e.g., how often does this problem occur over time?, in what otherwise non-problematic context does it occur?, is the problem cyclic in nature? Finally, consider gauges operating on behalf of different stakeholders in a system of systems deployment, who do not necessarily always have the same goals and requirements. Conflicts among what is monitored may not be a big problem, but it could become a big problem if contradictory repair plans were issued.

6.2 Controller Issues

Workflakes should be reimplemented without Cougaar, which is simply too heavy-weight. BPWS4J from IBM alphaworks, their implementation of the *de facto* BPEL4WS Web Services workflow language standard, might be suitable except the implementation details seem possibly too specific to Web Services. Another alternative is that U. Massachusetts be convinced to make its Little-JIL runtime engine externally available for research and education purposes.

The chosen (or newly developed) workflow engine must be specialized towards automated coordination of software entities, as opposed to the more conventional use of workflow to organize human activities. It would, as in Workflakes, be triggered by gauge outputs to choose a dynamic adaptation plan, then instantiate or recruit and then superintend an appropriate collection of effectors to enact the plan's tasks. The main area of new investigation would be constructing plans dynamically from predefined task elements, rather than selecting among "canned" plans with relatively minimal customization. Plan elements would probably be ordered using conventional preconditions and postconditions from AI planning, possibly also considering Perry's obligations for richer dependencies among tasks. The hardest part would likely be deriving contingency handling.

However, workflow is not necessarily the best technology for decision and control, as it is relatively weak with respect to decision aside from the micro level of exception handling. Workflow seems like the "obvious" approach to the Columbia PI, but that may be because she spent the first half of her career investigating software process workflow engines. Alternatives such as the various non-workflow software coordination languages and multi-agent coordination mechanisms should be explored.

Finally, it would be desirable to critically examine the presumption that decision and control are best united. Section 5.2 sketches a small experiment uniting Workflakes as the controller with CMU's architecture-based repair construction facility, but this was unfortunately rather cumbersome because both wanted to be "on top", invoking the other as a subroutine. The decision and control roles could potentially be subdivided further, e.g., into deciding there indeed is a problem (which to some extent gauges already do) vs. deciding what to do about it (architecture-based strategy) vs. how to go about it (workflow tactics).

Appendix A – WPI Subcontractor Final Report

A.1 Executive Summary

Columbia University and Worcester Polytechnic Institute (WPI) jointly proposed an architecture-based approach to run-time monitoring - i.e., continual validation - of the dynamic functional and extra-functional properties of component-based systems. Engineers would define the architecture of the target system in an architecture description language (ADL), which would then be inspected by our technology to semi-automatically insert software probes into component ports and actualized connector middleware/wrappers. These probes detect and report system events that cross component boundaries. Events are broadly construed including, but not limited to, procedure calls, I/O, memory, disk or network access, message passing, or hardware signals. Gauges can be integrated with automated decision facilities or directly displayed in a human-oriented GUI. (Un)desirable properties might optionally be specified to help identify temporal patterns that probabilistically foreshadow impending problems, and trigger preemptive reconfiguration through complementary continual coordination (and, indirectly, continual design). Probes could also insert anomalous events, for diagnostic purposes.

The probes provide the entry points to an orthogonal monitoring meta-architecture (superimposed on the target system's architecture), wherein the connectors operate as active connectors. The events generated by probes are converted to smart events represented in the eXtensible Markup Language (XML). Active connectors parse the XML streams, validate against the Document Type Definitions (DTDs), and transform and present as gauges according to eXtensible Stylesheet Language (XSL) style sheets. In addition to heading conventional meta-data (e.g., source host, timestamp), the markup tags indicate how to process these events to manage and update gauges, directly carrying the (mobile) code or describing where to find it. The meta-architecture is extensible and supports sophisticated gauges that may acquire, at run-time, XML processing modules for specific tagsets. This dynamism enables new gauges to be defined, and presented and acted upon, while a system is running - with no "down time" or significant reconfiguration solely to retrofit the monitoring infrastructure. It also neatly addresses components from independent sources, perhaps with their own built-in monitoring and diagnostic approaches, as well as loosely coupled systems with "protections" limiting event propagation, e.g., across firewalls.

A.2 Objectives

The DASADA paradigm shift offers a promising approach to adopt and leverage the proven functionality of existing software assets: continual design, continual coordination and continual validation. In this model, components are constructed, customized and evaluated before, during, and after system assembly and on-the-fly reassembly, to ensure that they can and do operate together with the rest of the system, and its current context, within the tolerated bounds. Continual validation is particularly essential for assured applications, since assurances that may have been met at initial system design time may not prove to be appropriate for field conditions that may be subject to rapid change while the system is running. Such applications typically cannot be "taken down" for long reengineering or enhancement cycles, but must be dynamically assembled in response to feedback from run-time gauges of functional and extra-functional system properties.

Two necessary bases for all three facets of dynamic assembly are: (1) being able to precisely determine and usefully specify the variation inherent in using "foreign" software components, connectors, and their configurations; and (2) being able to measure that software components, including middleware and other actualized connectors, fit and continue to fit together as system and context change, within functional and extra-functional tolerances permitted by the dynamically evolving requirements of critical software systems. The measurement probes must be insertable into legacy as well as new components and compositions and presentable to humans and automated decision algorithms in timely fashion as readily interpretable gauges, to prevent inappropriate system assemblies and trigger reassemblies promptly when needed.

However, it is simply impossible to include all possible probes and gauges, and appropriate mechanisms for interpreting gauge notifications, say, as customizations built into a system when it is designed or initially deployed: New components and platforms, or advancing technologies, may mandate or enable new metrics and feedback (or, as we advocate, feed-forward [SG96]). Thus we need a practical means for rapidly inserting and configuring new probes and gauges, and setting up their feed-forward loops, without significantly degrading system performance. Moreover, of course, measurement tasks and data must not themselves compromise high assurance, high dependability, and high adaptability properties. One implication is that such data and tasks should be accessed and applied only on a "need to know" basis, rather than swamping communication, computation and storage resources by dispersing every event throughout the distributed architecture – without knowledge of where it is "needed" (e.g., to maintain a particular gauge).

Columbia University and Worcester Polytechnic Institute (WPI) jointly proposed an architecture-based approach to dynamic gauging - i.e., continual validation - of the run-time functional and extra-functional properties of component-based systems. The target system's architecture definition assists in semi-automatic insertion of (software) *probes* into component ports and actualized connector middleware/wrappers. Probes detect and report system events crossing component boundaries. "Events" are broadly construed, including but not limited to procedure calls, I/O, memory, disk or network access, message passing, or hardware signals. Properties could be presented as binary (yes/no) *gauges*, or as sophisticated gauges providing detailed information about what went wrong and when (e.g., a partial match against a protocol modeled as a Finite-State Automata(FSA) CVACOP). Gauges can be routed to automated decision facilities and/or directly displayed in a human-oriented GUI. (Un)desirable properties might optionally be specified, to aid identification of temporal patterns that probabilistically foreshadow impending problems, and trigger pre-emptive reconfiguration through complementary continual coordination (and, indirectly, continual design). Probes could also insert anomalous events, for diagnostic purposes.

A.3 Approach

Our approach combined the following three efforts:

1. We attached "probes" to each component port, introducing before and after callbacks wrapped around each entry point, in the style of our active interfaces paradigm [Hei98b] - but now extended beyond procedure call to other connector types, including events, data accesses, linkages, streams, arbitrators, adaptors, and distributors (as in the taxonomy of Mehta *et al* [MMP99]). We developed as a deliverable the *dasada probe infrastructure* to be used as a basis for probing and monitoring the behavior of Java programs. The technology is language-independent, however, and can be ported to apply to other languages, such as C or C++. These probes rely on the Siena Internet-scale Event Notification Architecture to publish events for interested parties (gauges and others, as described in §2).

2. We introduced a system of "gauges" that receive information from the probes. Within the larger DASADA community, there were numerous research groups developing domain-specific gauges that were "pre-targeted" for particular scenarios. We focused on the broader, more generic class of gauges that would more likely be considered to be a *visualization engine*. The *SoftViz* laboratory is the resulting deliverable that receives probe information from the dasada probe infrastructure (§1) and produces dynamic visualizations of the behavior of the system as it happens. SoftViz is thus a more powerful tool than a static visualizer of code structure or one that allows a post-mortem playback of events. All events visualized by SoftViz are received from the dasada probe infrastructure and can be treated as "live" (naturally offset by the latency in receiving and processing the event).

3. The end-users of our monitoring systems will be guided and supported in their development of system-specific probes, gauges, gaugents, etc. by a Continual VAlidation COntrol Panel (CVACOP) customized for the target system's component model. For the deliverables, we developed the core infrastructure described in §1 as well as the means to develop and deploy probes on-the-fly, once the underlying code base is properly instrumented. Thus we serve to create both static support (i.e., the essence of probes, how they are deployed, installed, and activated) and dynamic support (i.e., the ability to generate and package probes as needed).

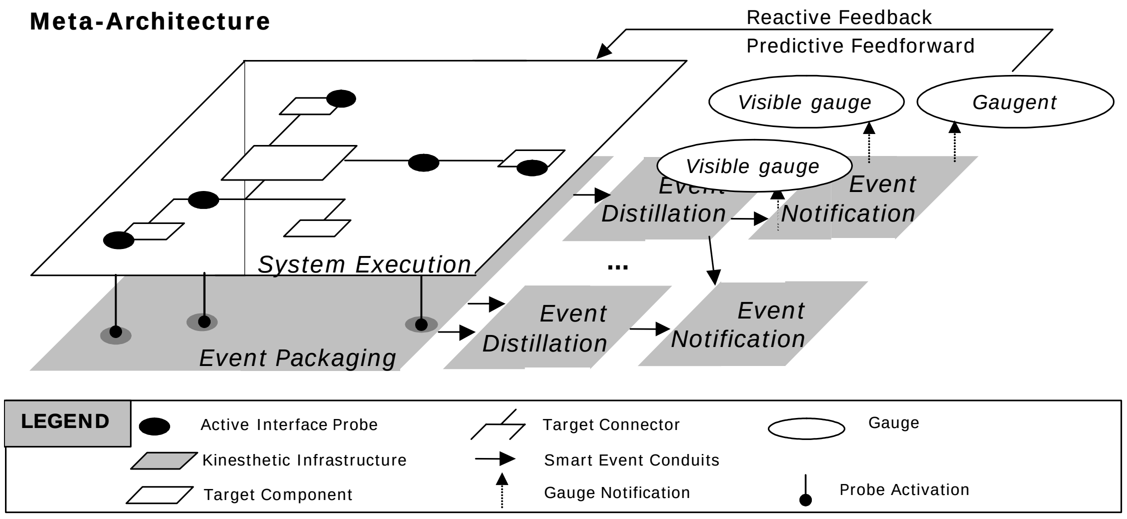


Figure A-1: WPI Meta-Architecture

A.3.1 Design

The design of the Kinesthetic Extreme (KX) meta-architecture is shown in Figure A-1. The three approaches of our research effort combine to form a powerful feedback/feed-forward system that enables the monitoring and reconfiguration of distributed component-based software systems. The probes (part 1 of our approach) are embedded within the target system using the AIDE technology (Deliverable **TD1** and **TD1.1**). Events are then delivered to waiting gauges over the Siena Event substrate. Gauges (part 2) receive these events and report status information to users; some gauges can be pre-configured to take action upon the observation of threshold values, in which case the reactive feedback loop can reconfigure or otherwise alter the execution of the underlying target system (Deliverable **TD4**). To enable rapid user acceptance of the probing technology, a comprehensive supporting infrastructure was developed (Deliverable **TD3** and part 3 of our approach). In this final report of the DASADA effort, we outline the primary deliverables for the WPI subcontract of the Columbia University prime contract.

A.4 Detailed Deliverables

The **KX** research and development effort consists of four major sub-projects: Active Interface Probing (AIP), Smart Event Active Connector Infrastructure (SEACI), Continual VAlidation COntrol Panel toolkit (CVACOP), and Reactive Gaugent Framework (RGF).

The deliverables shown below are drawn from the original technical proposal submitted jointly by Columbia University (prime) and WPI (subcontractor), and form the complete set of deliverables for subcontract WPI. Each task description (TD) is numbered as originally listed within the technical proposal:

- TD1: Active Interface Probing (AIP)
- TD1.1: Architectural Validation toolkit (AVT)
- TD3: Continual VAlidation COntrol Panel toolkit (CVACOP)
- TD4: Reactive Gaugent Framework (RGF)

Items from the original proposal not listed above (most notably the Smart Event Active Connector Infrastructure [SEACI]) are part of Columbia's deliverables (contact Dr. Gail E. Kaiser).

A.4.1 Active Interface Probing (TD1) and Architectural Validation Toolkit (TD1.1)

We developed an active interface development environment (AIDE) to aid developers in incorporating active interfaces into their programs. AIDE has a parsing tool that pre-processes a Java source file to embed the probes necessary for the gauge framework. Based on the ADAPT project, an NSF-funded effort (CCR-9733660), AIDE contains tools to automatically pre-process select methods under the guidance of a software engineer.

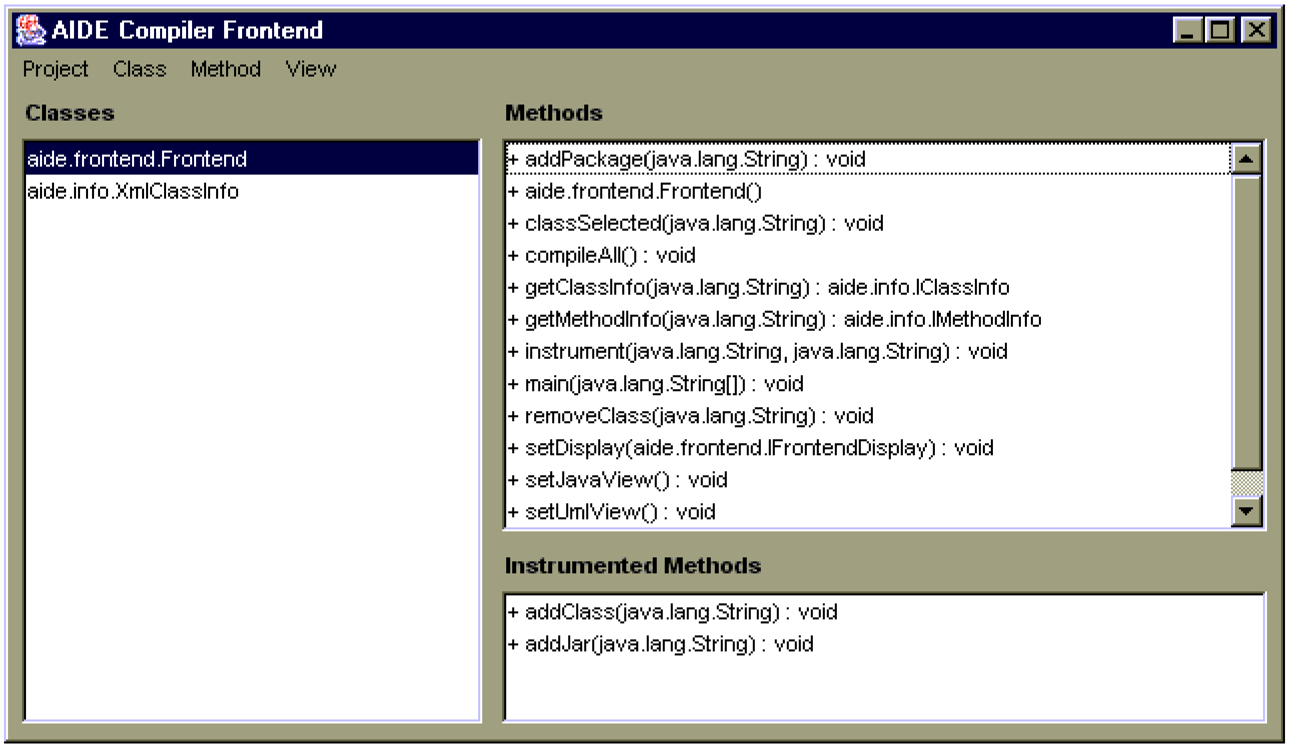
A.4.1.1 Concepts

We studied existing architecture description languages (ADLs) settling upon the use of ACME, as supported by AcmeLib [CMU99]. We investigated tools to exploit ADL specifications of target architectures to guide AIDE to automatically or semi-automatically insert software probes into component ports and actualized connector middleware/wrappers.

The concept of a probe is varied. For the purposes of AIDE, a *probe* can be attached to an *instrumentation point* within a target system; thus, the system must be pre-instrumented before it can receive a probe. When a method is instrumented for probing, its first statement will be a callback out to a possible probe (or set of probes); in addition, every exit point of the method is instrumented to insert a callback out to a probe immediately before the method is about to exit. The probe is essentially a method that will be invoked either *before* a method is to execute or just before it is about to exit (called the *after* probe). The *after*-probe has access to the return value, while both *before*- and *after*-probes are given the parameters to the original method. This probe model is inspired by earlier work done by the PI on adaptable software components (NSF grant CCR-9733660).

A.4.1.2 Results

We investigated the necessary capabilities of the probes themselves and how to insert probes into component ports for components whose source code is available and thus can be pre-processed. This remained the primary focus for the contract, because of the rich data available to probes for such target software. There were other available probing technologies from other DASADA researchers (most notably Balzer's instrumenting connectors [BG99] and ProbeMeister [Object Services and Consulting, Inc., www.objs.com/DASADA]). The events emitted from WPI's AIDE probes follow the same schema as events generated by other probing technologies.



aide.frontend.Frontend
aide.info.XmlClassInfo

+ addPackage(java.lang.String) : void
+ aide.frontend.Frontend()
+ classSelected(java.lang.String) : void
+ compileAll() : void
+ getClassInfo(java.lang.String) : aide.info.IClassInfo
+ getMethodInfo(java.lang.String) : aide.info.IMethodInfo
+ instrument(java.lang.String, java.lang.String) : void
+ main(java.lang.String[]) : void
+ removeClass(java.lang.String) : void
+ setDisplay(aide.frontend.IFrontendDisplay) : void
+ setJavaView() : void
+ setUmlView() : void

+ addClass(java.lang.String) : void
+ addJar(java.lang.String) : void

Figure A-2: AIDE Compiler Frontend.

Positive Results

The generic probes created by AIDE can be attached to any Java method of a class, including static methods. All parameters to the method are packaged and delivered to the probe for inspection. The probes are configured to work properly with the DASADA probe infrastructure and are thus unaware of the underlying event infrastructure for delivering the probe events.

Each probe is lightweight. When installed and inactive, the overhead is a single `if` statement per method call. When activated, the overhead is a Hashtable lookup (near constant time) and Hashtables are created for each required probe (so the space overhead is directly related to the number of probes installed).

Probes are packaged using the Java standard Java Archive (JAR) file format. Each probe is packaged with all supporting classes (if any) required to perform its operations. The

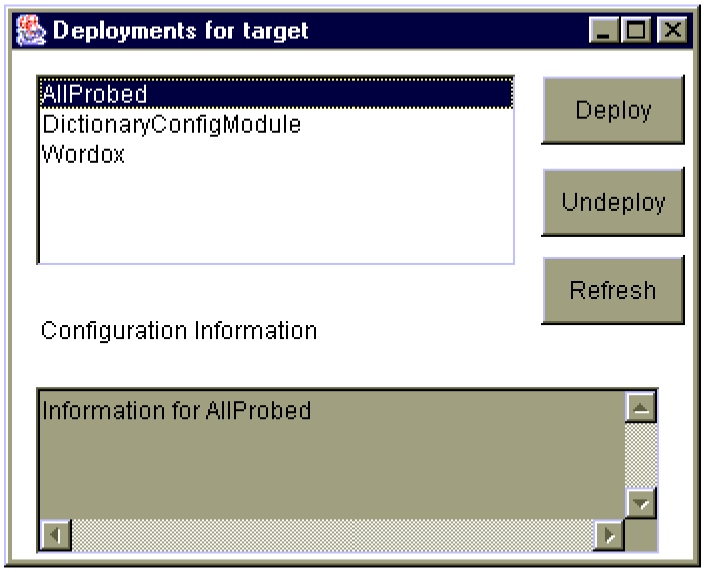
AIDE graphical user interface Compiler Frontend (see **Figure A-2**) enables users to select classes to be probed. AIDE records all choices so the user can easily alter the configuration of instrumented classes to which probes can be installed. The GUI Compiler Frontend enables users to select numerous classes and then, in a single operation, all selected classes (and their respective methods) are instrumented. All information about instrumented classes are stored in a set of XML meta-information files, for parsing by the AIDE Compiler Frontend, as well as the back end pre-processing engine that inserts the instrumentation. Thus, instrumenting can be viewed as simply one additional step in the compilation of the target system.

Other groups, most notably David Garlan of CMU, have used the AIDE infrastructure developed at WPI successfully. Probes form part of the internal mechanisms for joint work for Kaiser and Valetto.

Negative Results

AIDE still has some features that are not fully integrated into the probing environment. For example, Java enables static blocks of code to be executed whenever a Java class is loaded, and currently these cannot have probes attached to them.

The basic model of probing methods is both simple and powerful, but there are some weaknesses. Specifically, probes must be compiled separately from the target code base, yet the parameters to the probes use the types and classes developed within the target code base. To compile these probes, the user must set the CLASSPATH appropriately in order for Java's Classloader to work properly.



Information for AllProbed

Figure A-3: Probe Deployment

A.4.2 Continual Validation Control Panel Toolkit (TD3)

The Continual Validation Control Panel Toolkit (CVACOP) provides the vehicles needed to properly manage, reason about, and control a target component-based software system under the control of the KX system. CVACOP is built around a core infrastructure, called the *DASADA probe infrastructure*, which enables the run-time management of probes and their information. Thus, the "deliverable" for CVACOP is a set of Java code libraries that enable various probe- and gauge-development tools to properly function. The successful demonstrations of the DASADA probe infrastructure are included in the deliverables, and they include: (1) a small client/server game application used as a proof-of-concept; (2) a dynamic client/server communication system with clients that can be relocated dynamically based upon changes in server policy. This demonstration is provided under the Consolidated DASAD Infrastructure (CDI) deliverables.

A.4.2.1 Concepts

The first contribution of the dasada probe infrastructure is the underlying facility for extracting information from run-time systems. We examined the ways that probes can collect information from running systems and cataloged a number of existing technologies that could serve as probes. We explicitly described a number of probe types that can be implemented using Active Interfaces. Our prototype information relies on lightweight Java probes that are connected to instrumentation points created by the Active Interface Development Environment (AIDE).

A probe lifecycle is enabled by the dasada probe infrastructure. Probes are compiled and packaged into probe *configuration modules*, each of which represents a set of cohesive probes that must all be activated, or deactivated, as a unit. Probes are deployed in these units, and then installed and activated as appropriate. **Figure A-3** shows the user interface whereby users can register and deploy probe configurations.

The second contribution was the development of a probe run-time infrastructure that can be used to deploy and manage probes at run time. We described a high level design that would allow for implementation of the run-time infrastructure for the various probe mechanisms. We provided an implementation and demonstration of the design for Active Interfaces. The run-time infrastructure supports the monitoring of constructors, method calls, and static class methods. Each probe module is configured for a specific target system, and the deployment occurs using standard Hyper-text Transfer Protocol (HTTP) to transfer probe modules and The Scalable Internet Event Notification Architecture (SIENA) for all control messages.

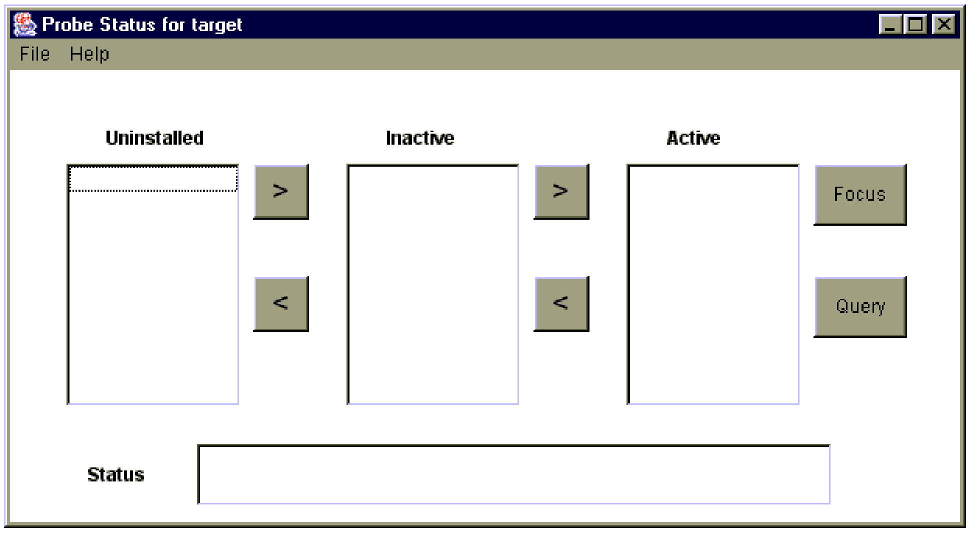


Figure A-4: Probe Lifecycle Management.

Once a probe is *deployed*, the configuration module (a Java Archive File) is downloaded to prepare it for instantiation and it is in the *uninstalled* state. It can be transformed into two states. An *installed* probe does no processing but can be *activated* upon request. Once active, a probe emits the events for which it has been created. Probes can rapidly be activated and deactivated as the system operates, allowing engineers to fine-tune their troubleshooting to focus in on problem areas of the system. The graphical user interface for manipulating the state of probes, called the *Probe Configuration Window*, is shown in **Figure A-4**.

The final contribution of the DASADA probe infrastructure is the DASADA shell within which the target software system is deployed. The shell manages the set of probes deployed for a target system, responding to control messages from the probe configuration window. Because HTTP is used for deploying probes, the shell can be remote from the user machine issuing the requests. The shell further manages the set of instantiated probes and controls their communication for emitting events. The shell footprint is small and each element within the distributed target system must operate within the context of a shell.

The DASADA probe infrastructure comes with a sample demonstration environment of a client/server network game. Both the client and the server in this demonstration can be probed, showing the flexibility and power of the approach.

A.4.2.2 Results

Positive Results

The DASADA shell is the leveraging technology that enables instrumented Java classes to deploy and activate probes. It is a lightweight front-end that enables any stand-alone

Java application to be provided with an alternative means of launching the application, one that is DASADA-aware. Once the shell is activated, it is responsible for listening to underlying control messages that are sent over the SIENA substrate. The shell enabled distributed component-based systems to be monitored in a consistent manner, because all events are broadcast to awaiting listeners over the SIENA substrate.

By using existing standards for packaging (JAR) and deployment (HTTP), probe management is simplified. The graphical interfaces shown in Figure 3 and Figure 4 provide the capability to enable managers to monitor and manage all active probe modules in the system. Each system being observed is assigned a “unique system ID” to ensure that the appropriate control messages and probe events are delivered to the appropriate listeners.

The DASADA shell persistently stores the set of activate probes, the set of installed (but inactive) probes, and the set of deployed (but uninstalled) probes. In doing so, start-up time is greatly reduced, and the system is able to quickly recover the probe information from failures. The lifecycle for probes is defined by a state machine of allowable transitions and thus it could be extended (although this was not a primary requirement of the supplied deliverables).

Negative Results

The use of SIENA for transmitting events greatly enabled the building of the prototype systems. However, there is overhead that becomes apparent when the probed events occur rapidly, that is, when events are emitted faster than about five per second. We ensured that the probe infrastructure is as independent as possible from the underlying substrate, so one could envision a follow-up project that incorporates a more efficient event engine.

There will invariably be perturbations in the target system when probes are invoked. The overhead is impossible to measure, because probes are generic and customized to emit events of interest to the probe. We only observed performance slow-downs, but it is theoretically possible that the probes could introduce race conditions, especially when multi-threaded programs are being probed.

A.4.3 Reactive Gaugent Framework (TD4)

The Reactive Gaugent Framework (RGF) was targeted to apply SEACI (a deliverable from Columbia University) to the feed-forward aspect of continual coordination (in additional to continual validation, the primary topic of this proposed effort). Because RGF needs to be contextualized for each domain to which it is applied, we provide prototyping facilities to DASADA contractors focusing on continual coordination for specific domains. The initial target domain is a reactive client/server system providing relocation services to clients (i.e., based on server policies, a client can be redirected to another server at a different host or port).

A.4.3.1 Concepts

Software systems can be expressed in terms of an architecture consisting of interacting components. This architecture for a system is abstract and often ideal; its implementation is often imperfect and does not always follow the architecture's structure. This inconsistency prohibits a straightforward mapping of changes in architecture to changes in implementation. The CDI prototype for the RGF aimed to develop a standard means of mapping standard architectural changes to non-standard implementation changes. As such, it is the most powerful evidence that the overall goals of the KX project, and the entire DASADA project, is valid.

Several architecture description languages (ADLs) have been developed over the years, typically aimed at solving a specific problem in a particular domain. When describing an architecture, one can identify two distinct sets of definitions:

- *Architectural state* depicts specific instances of components and specific interconnections for a give system at a give time. It is much like a snapshot of a system's state at an architectural level.
- *Architectural style* describes the vocabulary of component types and other architectural elements that can be used to build a particular architectural state. It can be considered the template from which allowed states can be constituted.

Similar systems within the DASADA community have been built, notably Armani and Tailor from CMI [M01][SG02]. The CDI prototype we developed fully integrates the dasada probe infrastructure from § 5.2 and the AIDE technology from § 5.1 to monitor the run-time behavior of systems.

A.4.3.2 Results

This project develop an architecture to implementation mapping that (1) bridges a decision (i.e., an architectural change) and a implementation in one direction and; (2) updates the architectural state to reflect implementation changes. We standardized the description of architectural changes by adopting a vocabulary of architectural events and a set of rules and procedures for using these events. Architectural events (AEs) manipulate an architectural state to produce a new, modified architectural state. Desired architectural changes are

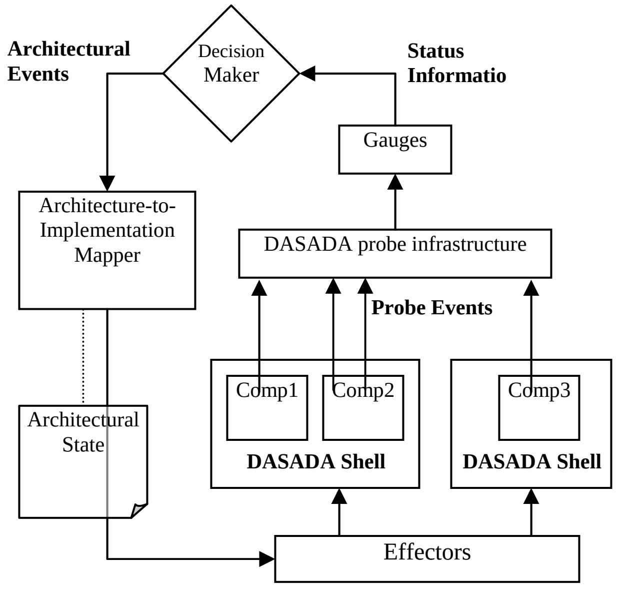


Figure A-5: CDI Architecture.

modeled using a series of primitive events (i.e., createComponent, attachConnector). These events are then projected onto the running system through the aide of run-time *effectors* that are responsible for understanding how to carry out the desired low-level effect given a high-level change. Full details on the architectural events and their composition can be found in [ERZ03].

Positive Results

The prototype system is able to translate observed behaviors from the emitted probe events into architectural reconfigurations that can be handled in a domain-specific way by effectors tailored for the targeted software system.

We defined a standardized interface for effectors and the architectural state, allowing each to be configured to the appropriate target system or architectural definition language. Our target prototype uses the ACME ADL to represent state information, to enhance the interoperability with DASADA tools from other researchers. The prototype, called Consolidated Dasada Infrastructure (CDI), is described in [ERZ03]. A demo of the CDI is provided, with scripts and other supporting programs tailored for (1) Windows platforms; and (2) Unix or Linux. The directions for installation and running the demonstrations can be found within the CDI deliverables.

We expect this to be more dependent on the target component framework, if any, than pure monitoring. Further, gaugents will generally be very specific to the target domain, so we provided prototyping facilities to DASADA contractors focusing on continual coordination for specific domains or target systems.

Negative Results

We had to make one change to the dasada probe infrastructure to support the need for effectors to observe the sudden disappearance of a process within the DASADA shell. The change was slight, however, and is the only instance where we were unable to cleanly separate the dasada probe infrastructure and CDI.

While we have a working system, the *decision maker* as shown in **Figure A-5** is pre-programmed to react to specific events of interest. There needs to be more work (outside the scope of this effort) to make this component an effective and "intelligent" agent for recommending changes to the underlying run-time system.

A.5 Summary

This is the final report for WPI subcontractor for contract F30602-00-2-0611, Columbia University Prime contractor (Gail E. Kaiser). The contact for WPI is George Heineman (heineman@cs.wpi.edu) and all deliverables can be retrieved (including this final report) from www.cs.wpi.edu/~heineman/dasada.

A.6 References

[BG99] R. Balzer and N. Goldman. Mediating Connectors. 19th IEEE international Conference on Distributed Computing Systems Workshop, June 1999.

[C03] P. Calnan, Extensible Transformation and Compiler Technology, WPI Master's Thesis, May 2003.

[CMU99] Acme Tools. Carnegie Mellon University School f Computer Science, June 1999. http://www.cs.cmu.edu/~acme/acme_downloads.html.

[ERZ03] A. L. Elliot, O. Rekutin, and C. A. Znoj, Consolidated Dasada Infrastructure, Major Qualifying Project, WPI, Report No. 03C035M, www.wpi.edu/cgi-bin/display_project?03C035M:58:1521, May 2003.

[G01] P. Gill, Probing for a Continual Validation Prototype, WPI Master's Thesis, Aug. 2001.

[Hei98b] G. T. Heineman. A Model for Designing Adaptable Software Components. 22nd Annual International Computer Software and Applications Conference, Aug. 1998.

[K03] B. Kurtz, SoftViz: A Runtime Software Visualization Environment, WPI Master's Thesis, May 2003.

[MMP99] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. University of Southern California, Center for Software Engineering, Nov. 1999, USCCSE99-529.

[M01] R. T. Monroe. Capturing software architecture design expertise with Armani. Tech. Rep. CMU-CS-98-163, Carnegie Mellon University School of Computer Science, Jan. 2001.

[SG96] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, April 1996.

[SG02] B. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. Proceedings, Fourteenth International Conference on Software Engineering and Knowledge Engineering, July 2002.